MITOCW | Lec-06

SPEAKER 1: It was about 1963 when a noted philosopher here at MIT, named Hubert Dreyfus-- Hubert Dreyfus wrote a paper in about 1963 in which he had a heading titled, "Computers Can't Play Chess." Of course, he was subsequently invited over to the artificial intelligence laboratory to play the Greenblatt chess machine.

And, of course, he lost.

Whereupon Seymour Pavitt wrote a rebuttal to Dreyfus' famous paper, which had a subject heading, "Dreyfus Can't Play Chess Either." But in a strange sense, Dreyfus might have been right and would have been right if he were to have said computers can't play chess the way humans play chess yet.

In any case, around about 1968 a chess master named David Levy bet noted founder of artificial intelligence John McCarthy that no computer would beat the world champion within 10 years.

And five years later, McCarthy gave up, because it had already become clear that no computer would win in a way that McCarthy wanted it to win, that is to say by playing chess the way humans play chess.

But then 20 years after that in 1997, Deep Blue beat the world champion, and chess suddenly became uninteresting.

But we're going to talk about games today, because there are elements of game-play that do model some of the things that go on in our head.

And if they don't model things that go on in our head, they do model some kind of intelligence.

And if we're to have a general understanding of what intelligence is all about, we have to understand that kind of intelligence, too.

So, we'll start out by talking about various ways that we might design a computer program to play a game like chess.

And we'll conclude by talking a little bit about what Deep Blue adds to the mix other than tremendous speed.

So, that's our agenda.

By the end of the hour, you'll understand and be able to write your own Deep Blue if you feel like it.

First, we want to talk about how it might be possible for a computer to play chess.

Let's talk about several approaches that might be possible.

Approach number one is that the machine might make a description of the board the same way a human would;

talk about pawn structure, King safety, whether it's a good time to castle, that sort of thing.

So, it would be analysis and perhaps some strategy mixed up with some tactics.

And all that would get mixed up and, finally, result in some kind of move.

If this is the game board, the next thing to do would be determined by some process like that.

And the trouble is no one knows how to do it.

And so in that sense, Dreyfus is right.

None the game playing programs today incorporate any of that kind of stuff.

And since nobody knows how to do that, we can't talk about it.

So we can talk about other ways, though, that we might try.

For example, we can have if-then rules.

How would that work?

That would work this way.

You look at the board, represented by this node here, and you say, well, if it's possible to move the Queen pawn forward by one, then do that.

So, it doesn't do any of evaluation of the board.

It doesn't try anything.

It just says let me look at the board and select a move on that basis.

So, that would be a way of approaching a game situation like this.

Here's the situation.

Here are the possible moves.

And one is selected on the basis of an if-then rule like so.

And nobody can make a very strong chess player that works like that.

Curiously enough, someone has made a pretty good checkers playing program that works like that.

It checks to see what moves are available on the board, ranks them, and picks the highest one available.

But, in general, that's not a very good approach.

It's not very powerful.

You couldn't make it-- well, when I say, couldn't, it means I can't think of any way that you could make a strong chess playing program that way.

So, the third way to do this is to look ahead and evaluate.

What that means is you look ahead like so.

You see all the possible consequences of moves, and you say, which of these board situations is best for me?

So, that would be an approach that comes in here like so and says, which one of those three situations is best?

And to do that, we have to have some way of evaluating the situation deciding which of those is best.

Now, I want to do a little, brief aside, because I want to talk about the mechanisms that are popularly used to do that kind of evaluation.

In the end, there are lots of features of the chessboard.

Let's call them f1, f2, and so on.

And we might form some function of those features.

And that, overall, is called the static value.

So, it's static because you're not exploring any consequences of what might happen.

You're just looking at the board as it is, checking the King's safety, checking the pawn structure.

Each of those produces a number fed into this function, out comes a value.

And that is a value of the board seen from your perspective.

Now, normally, this function, g, is reduced to a linear polynomial.

So, in the end, the most popular kind of way of forming a static value is to take f1, multiply it times some constant, c1, add c2, multiply it times f2.

And that is a linear scoring polynomial.

So, we could use that function to produce numbers from each of these things and then pick the highest number.

And that would be a way of playing the game.

Actually, a scoring polynomial is a little bit more than we need.

Because all we really need is a method that looks at those three boards and says, I like this one best.

It doesn't have to rank them.

It doesn't have to give them numbers.

All it has to do is say which one it likes best.

So, one way of doing that is to use a linear scoring polynomial.

But it's not the only way of doing that.

So, that's number two and number three.

But now what else might we do?

Well, if we reflect back on some of the searches we talked about, what's the base case against which everything else is compared much the way of doing search that doesn't require any intelligence, just brute force?

We could use the British Museum algorithm and simply evaluate the entire tree of possibilities; I move, you move, I move, you move, all the way down to-- what?-- maybe 100, 50 moves.

You do 50 things.

I do 50 things.

So, before we can decide if that's a good idea or not, we probably ought to develop some vocabulary.

So, consider this tree of moves.

There will be some number of choices considered at each level.

And there will be some number of levels.

So, the standard language for this as we call this the branching factor.

And in this particular case, b is equal to 3.

This is the depth of the tree.

And, in this case, d is two.

So, now that produces a certain number of terminal or leaf nodes.

How many of those are there?

Well, that's pretty simple computation.

It's just b to the d.

Right, Christopher, b to the d?

So, if you have b to the d at this level, you have one.

b to the d at this level, you have b.

b to the d at this level, you have [? d ?] squared.

So, b to the d, in this particular case, is 9.

So, now we can use this vocabulary that we've developed to talk about whether it's reasonable to just do the British Museum algorithm, be done with it, forget about chess, and go home.

Well, let's see.

It's pretty deep down there.

If we think about chess, and we think about a standard game which each person does 50 things, that gives a d about 100.

And if you think about the branching factor in chess, it's generally presumed to be, depending on the stage of the game and so on and so forth, it varies, but it might average around 14 or 15.

If it were just 10, that would be 10 to the 100th.

But it's a little more than that, because the branching factor is more than 10.

So, in the end, it looks like, according to Claude Shannon, there are about 10 to the 120th leaf nodes down there.

And if you're going to go to a British Museum treatment of this tree, you'd have to do 10 to the 120th static evaluations down there at the bottom if you're going to see which one of the moves is best at the top.

Is that a reasonable number?

It didn't used to seem practicable.

It used to seem impossible.

But now we've got cloud computing and everything.

And maybe we could actually do that, right?

What do you think, Vanessa, can you do that, get enough computers going in the cloud?

No?

You're not sure?

Should we work it out?

Let's work it out.

I'll need some help, especially from any of you who are studying cosmology.

So, we'll start with how many atoms are there in the universe?

Volunteers?

10 to the-- SPEAKER 2: 10 to the 38th?

SPEAKER 1: No, no, 10 to the 38th has been offered.

That's why it's way too low.

The last time I looked, it was about 10 to the 80th atoms in the universe.

The next thing I'd like to know is how many seconds are there in a year?

It's a good number have memorized.

That number is approximately pi times 10 to the seventh.

So, how many nanoseconds in a second?

That gives us 10 to the ninth.

At last, how many years are there in the history of the universe?

SPEAKER 3: [INAUDIBLE].

14.7 billion.

SPEAKER 1: She offers something on the order of 10 billion, maybe 14 billion.

But we'll say 10 billion to make our calculation simple.

That's 10 to the 10th years.

If we will add that up, 80, 90, plus 16, that's 10 to the 106th nanoseconds in the history of the universe.

Multiply it times the number of atoms in the universe.

So, if all of the atoms in the universe were doing static evaluations at nanosecond speeds since the beginning of the Big Bang, we'd still be 14 orders of magnitudes short.

So, it'd be a pretty good cloud.

It would have to harness together lots of universes.

So, the British Museum algorithm is not going to work.

No good.

So, what we're going to have to do is we're going to have to put some things together and hope for the best.

So, the fifth way is the way we're actually going to do it.

And what we're going to do is we're going to look ahead, not just one level, but as far as possible.

We consider, not only the situation that we've developed here, but we'll try to push that out as far as we can and look at these static values of the leaf nodes down here and somehow use that as a way of playing the game.

So, that is number five.

And number four is going all the way down there.

And this, in the end, is all that we can do.

This idea is multiply invented most notably by Claude Shannon and also by Alan Turing, who, I found out from a friend of mine, spent a lot a lunch time conversations talking with each other about how a computer might play chess against the future when there would be computers.

So, Donald, Mickey and Alan Turing also invented this over lunch while they were taking some time off from cracking the German codes.

Well, what is the method?

I want to illustrate the method with the simplest possible tree.

So, we're going to have a branching factor of 2 not 14.

And we're going to have a depth of 2 not something highly serious.

Here's the game tree.

And there are going to be some numbers down here at the bottom.

And these are going to be the value of the board from the perspective of the player at the top.

Let us say that the player at the top would like to drive the play as much as possible toward the big numbers.

So, we're going to call that player the maximizing player.

He would like to get over here to the 8, because that's the biggest number.

There's another player, his opponent, which we'll call the minimizing player.

And he's hoping that the play will go down to the board situation that's as small as possible.

Because his view is the opposite of the maximizing player, hence the name minimax.

But how does it work?

Do you see which way the play is going to go?

How do you decide which way the play is going to go?

Well, it's not obvious at a glance.

Do you see which way it's going to go?

It's not obvious to the glance.

But if we do more than a glance, if we look at the situation from the perspective of the minimizing player here at the middle level, it's pretty clear that if the minimizing player finds himself in that situation, he's going to choose to go that way.

And so the value of this situation, from the perspective of the minimizing player, is 2.

He'd never go over there to the 7.

Similarly, if the minimizing player is over here with a choice between going toward a 1 or toward an 8, he'll obviously go toward a 1.

And so the value of that board situation, from the perspective of the minimizing player, is 1.

Now, we've taken the scores down here at the bottom of the tree, and we back them up one level.

And you see how we can just keep doing this?

Now the maximizing player can see that if he goes to the left, he gets a score of 2.

If he goes to the right, he only gets a score of 1.

So, he's going to go to the left.

So, overall, then, the maximizing player is going to have a 2 as the perceived value of that situation there at the top.

That's the minimax algorithm.

It's very simple.

You go down to the bottom of the tree, you compute static values, you back them up level by level, and then you decide where to go.

And in this particular situation, the maximizer goes to the left.

And the minimizer goes to the left, too, so the play ends up here, far short of the 8 that the maximizer wanted and less than the 1 that the minimizer wanted.

But this is an adversarial game.

You're competing with each other.

So, you don't expect to get what you want, right?

So, maybe we ought to see if we can make that work.

There's a game tree.

Do you see how it goes?

Let's see if the system can figure it out.

There it goes, crawling its way through the tree.

This is a branching factor of 2, just like our sample, but now four levels.

You can see that it's got quite a lot of work to do.

That's 2 to the fourth, one, two, three, four, 2 to the fourth, 16 static evaluations to do.

So, it found the answer.

But it's a lot of work.

We could get a new tree and restart it, maybe speed it up.

There is goes down that way, get a new tree.

Those are just random numbers.

So, each time it's going to find a different path through the tree according to the numbers that it's generated.

Now, 16 isn't bad.

But if you get down there around 10 levels deep and your branching factor is 14, well, we know those numbers get pretty awful pretty bad, because the number of static evaluations to do down there at the bottom goes as b to the d.

It's exponential.

And time has shown, if you get down about seven or eight levels, you're a jerk.

And if you get down about 15 or 16 levels, you beat the world champion.

So, you'd like to get as far down in the tree as possible.

Because when you get as far down into the tree as possible, what happens is as these that these crude measures of bored quality begin to clarify.

And, in fact, when you get far enough, the only thing that really counts is piece count, one of those features.

If you get far enough, piece count and a few other things will give you a pretty good idea of what to do if you get far enough.

But getting far enough can be a problem.

So, we want to do everything we can to get as far as possible.

We want to pull out every trick we can find to get as far as possible.

Now, you remember when we talked about branching down, we knew that there were some things that we could do that would cut off whole portions of the search tree.

So, what we'd like to do is find something analogous to this world of games, so we cut off whole portions of this search tree, so we don't have to look at those static values.

What I want to do is I want to come back and redo this thing.

But this time, I'm going to compute the static values one at a time.

I've got the same structure in the tree.

And just as before, I'm going to assume that the top player wants to go toward the maximum values, and the next player wants to go toward the minimum values.

But none of the static values have been computed yet.

So, I better start computing them.

That's the first one I find, 2.

Now, as soon as I see that 2, as soon as the minimizer sees that 2, the minimizer knows that the value of this node can't be any greater than 2.

Because he'll always choose to go down this way if this branch produces a bigger number.

So, we can say that the minimizer is assured already that the score there will be equal to or less than 2.

Now, we go over and compute the next number.

There's a 7.

Now, I know this is exactly equal to 2, because he'll never go down toward a 7.

As soon as the minimizer says equal to 2, the maximizer says, OK, I can do equal to or greater than 2.

One, minimizer says equal to or less than 1.

Now what?

Did you prepare those 2 numbers?

The maximizer knows that if he goes down here, he can't do better than 1.

He already knows if he goes over here, he an get a 2.

It's as if this branch doesn't even exist.

Because the maximizer would never choose to go down there.

So, you have to see that.

This is the important essence of the notion the alpha-beta algorithm, which is a layering on top of minimax that cuts off large sections of the search tree.

So, one more time.

We've developed a situation so we know that the maximizer gets a 2 going down to the left, and he sees that if he goes down to the right, he can't do better than 1.

So, he says to himself, it's as if that branch doesn't exist and the overall score is 2.

And it doesn't matter what that static value is.

It can be 8, as it was, it can be plus 1,000.

It doesn't matter.

It can be minus 1,000.

Or it could be plus infinity or minus infinity.

It doesn't matter, because the maximizer will always go the other way.

So, that's the alpha-beta algorithm.

Can you guess why it's called the alpha-beta algorithm?

Well, because in the algorithm there are two parameters, alpha and beta.

So, it's important to understand that alpha-beta is not an alternative to minimax.

It's minimax with a flourish.

It's something layered on top like we layered things on top of branch and bound to make it more efficient.

We layer stuff on top of minimax to make it more efficient.

As you say to me, well, that's a pretty easy example.

And it is.

So, let's try a little bit more complex one.

This is just to see if I can do it without screwing up.

The reason I do one that's complex is not just to show how tough I am in front of a large audience.

But, rather, there's certain points of interest that only occur in a tree of depth four or greater.

That's the reason for this example.

But work with me and let's see if we can work our way through it.

What I'm going to do is I'll circle the numbers that we actually have to compute.

So, we actually have to compute 8.

As soon as we do that, the minimizer knows that that node is going to have a score of equal to or less than 8 without looking at anything else.

Then, he looks at 7.

So, that's equal to 7.

Because the minimizer will clearly go to the right.

As soon as that is determined, then the maximizer knows that the score here is equal to or greater than 8.

Now, we evaluate the 3.

The minimizer knows equal to or less than 3.

SPEAKER 4: [INAUDIBLE].

SPEAKER 1: Oh, sorry, the minimizer at 7, yeah.

OK, now what happens?

Well, let's see, the maximizer gets a 7 going that way.

He can't do better than 3 going that way, so we got another one of these cut off situations.

It's as if this branch doesn't even exist.

So, this static evaluation need not be made.

And now we know that that's not merely equal to or greater than 7, but exactly equal to 7.

And we can push that number back up.

That becomes equal to or less than 7.

OK, are you with me so far?

Let's get over to the other side of the tree as quickly as possible.

So, there's a 9, equal to or less than 9, 8 equal to 8, push the 8 up equal or greater than 8.

The minimizer can go down this way and get a 7.

He'll certainly never go that way where the maximizer can get an 8.

Once again, we've got a cut off.

And if this branch didn't exist, then that means that these static evaluations don't have to be made.

And this value is now exactly 7.

But there's one more thing to note here.

And that is that not only do we not have to make these static evaluations down here, but we don't even have to generate these moves.

So, we save two ways, both on static evaluation and on move generation.

This is a real winner, this alpha-beta thing, because it saves as enormous amount of computation.

Well, we're on the way now.

The maximizer up here is guaranteed equal to or greater than 7.

Has anyone found the winning media move yet?

Is it to the left?

I know that we better keep going, because we want to trust any oracles.

So, let's see.

There's a 1.

We've calculated that.

The minimizer can be guaranteed equal to or less than 1 at that particular point.

Think about that for a while.

At the top, the maximizer knows he can go left and get a 7.

the minimizer, if the play ever gets here, can ensure that he's going to drive the situation to a board number that's 1.

So, the question is will the maximizer ever permit that to happen?

And the answer is surely not.

So, over here in the development of this side of the tree, we're always comparing numbers at adjacent levels in the tree.

But here's a situation where we're comparing numbers that are separated from each other in the tree.

And we still concluded that no further examination of this node makes any sense at all.

This is called deep cut off.

And that means that this whole branch here might as well not exist, and we won't have to compute that static value.

All right?

So, it looks-- you have this stare of disbelief, which is perfectly normal.

I have to reconvince myself every time that this actually works.

But when you think your way through it, it is clear that these computations that I've x-ed out don't have to be made.

So, let's carry on and see if we can complete this equal to or less than 8, equal to 8, equal to 8-- because the other branch doesn't even exist-- equal to or less than 8.

And we compare these two numbers, do we keep going?

Yes, we keep going.

Because maybe the maximizer can go to the right and actually get to that 8.

So, we have to go over here and keep working away.

There's a nine, equal to or less than 9, another 9 equal to 9.

Push that number up equal to or greater than 9.

The minimizer gets an 8 going this way.

The maximizer is insured of getting a 9 going that way.

So, once again, we've got a cut off situation.

It's as if this doesn't exist.

Those static evaluations are not made.

This move generation is not made and computation is saved.

So, let's see if we can do better on this very example using this alpha-beta idea.

I'll slow it down a little bit and change the search type to minimax with alpha-beta.

We see two numbers on each of those nodes now, guess what they're called.

We already know.

They're alpha and beta.

So, what's going to happen is the algorithm proceeds through trees that those numbers are going to shrink wrap themselves around the situation.

So, we'll start that up.

Two static evaluations were not made.

Let's try a new tree.

Two different ones were not made.

A new tree, still again, two different ones not made.

Let's see what happens when we use the classroom example, the one I did up there.

Let's make sure that I didn't screw it up.

I'll slow that down to 1.

2, same answer.

So, you probably didn't realize it at the start.

Who could?

In fact, the play goes down that way, over this way, down that way, and ultimately to the 8, which is not the biggest number.

And it's not the smallest number.

It's the compromised number that's arrived at virtue of the fact that this is an adversarial situation.

So, you say to me, how much energy, how much work do you actually saved by doing this?

Well, it is the case that in the optimal situation, if everything is ordered right, if God has come down and arranged your tree in just the right way, then the approximate amount of work you need to do, the approximate number of static evaluations performed, is approximately equal to 2 times b to the d over 2.

We don't care about this 2.

We care a whole lot about that 2.

That's the amount of work that's done.

It's b to the d over 2, instead of b to d.

What's that mean?

Suppose that without this idea, I can go down seven levels.

How far can I go down with this idea?

14 levels.

So, it's the difference between a jerk and a world champion.

So, that, however, is only in the optimal case when God has arranged things just right.

But in practical situations, practical game situations, it appears to be the case, experimentally, that the actual number is close to this approximation for optimal arrangements.

So, you'd never not want to use alpha-beta.

It saves an amazing amount of time.

You could look at it another way.

Suppose you go down the same number of levels, how much less work do you have to do?

Well, quite a bit.

The square root [INAUDIBLE], right?

That's another way of looking at how it works.

So, we could go home at this point except for one problem, and that is that we pretended that the branching factor is always the same.

But, in fact, the branching factor will vary with the game state and will vary with the game.

So, you can calculate how much computing you can do in two minutes, or however much time you have for an average move.

And then you could say, how deep can I go?

And you won't know for sure, because it depends on the game.

So, in the earlier days of game-playing programs, the game-playing program left a lot of computation on the table, because it would make a decision in three seconds.

And it might have made a much different move if it used all the competition it had available.

Alternatively, it might be grinding away, and after two minutes was consumed.

It had no move and just did something random.

That's not very good.

But that's what the early game-playing program's did, because no one knew how deep they could go.

So, let's have a look at the situation here and say, well, here's a game tree.

It's a binary game tree.

That's level 0.

That's level 1.

This is level d minus 1.

And this is level d.

So, down here you have a situation that looks like this.

And I left all the game tree out in between .

So, how many leaf nodes are there down here?

b to the d, right?

Oh, I'm going to forget about alpha alpha-beta for a moment.

As we did when we looked at some of those optimal searches, we're going to add these things one at a time.

So, forget about alpha-beta, assume we're just doing straight minimax.

In that case, we would have to calculate all the static values down here at the bottom.

And there are b to d of those.

How many are there at this next level up?

Well, that must be b to the d minus 1.

How many fewer nodes are there at the second to the last, the penultimate level, relative to the final level?

Well, 1 over b, right?

So, if I'm concerned about not getting all the way through these calculations at the d level, I can give myself an insurance policy by calculating out what the answer would be if I only went down to the d minus 1th level.

Do you get that insurance policy?

Let's say the branching factor is 10, how much does that insurance policy cost me?

10% of my competition.

Because I can do this calculation and have a move in hand here at level d minus 1 for only 1/10 of the amount of the computation that's required to figure out what I would do if I go all the way down to the base level.

OK, is that clear?

So this idea is extremely important in its general form.

But we haven't quite got there yet, because what if the branching factor turns out to be really big and we can't get through this level either?

What should we do to make sure that we still have a good move?

SPEAKER 5: [INAUDIBLE].

SPEAKER 1: Right, we can do it at the b minus 2 level.

So, that would be up here.

And at that level, the amount of computation would be b to the d minus 2.

So, now we've added 10% plus 10% of that.

And our knee jerk is begin to form, right?

What are we going to do in the end to make sure that no matter what we've got a move?

CHRISTOPHER: Start from the very first-- SPEAKER 1: Correct, what's that, Christopher?

CHRISTOPHER: Start from the very first level?

SPEAKER 1: Start from the very first level and give our self an insurance policy for every level we try to calculate.

But that might be real costly.

So, we better figure out if this is going to be too big of an expense to bear.

So, let's see, if we do what Christopher suggests, then the amount of computation we need in our insurance policy is going to be equal 1-- we're going to do it up here at this level, 2, even though we don't need it, just to make everything work out easy.

1 plus b, that's getting or insurance policy down here at this first level.

And we're going to add b squared all the way down to b to d minus 1.

That's how much we're going to spend getting an insurance policy at every level.

I wished that some of that high school algebra, right?

Let's just do it for fun.

Oh, unfortunate choice of variable names.

bs is equal to-- oh, we're going to multiply all those by b.

Now, we'll subtract the first one from the second one, which tells us that the amount of calculation needed for our insurance policy is equal to b to the d minus 1 over b minus 1.

Is that a big number?

We could do a little algebra on that and say that b to the d is a huge number.

So, that minus one doesn't count.

And B is probably 10 to 15.

So, b minus 1 is, essentially, equal to b.

So, that's approximately equal b to the d minus 1.

So, with an approximation factored in, the amount of computation needed to do insurance policies at every level is not much different from the amount of computation needed to get an insurance policy at just one level, the penultimate one.

So, this idea is called progressive deepening.

And now we can visit our gold star idea list and see how these things match up with that.

First of all, the dead horse principle comes to the fore when we talk about alpha-beta.

Because we know with alpha-beta that we can get rid of a whole lot of the tree and not do static evaluation, not even do move generation.

That's the dead horse we don't want to beat.

There's no point in doing that calculation, because it can't figure into the answer.

The development of the progressive deepening idea, I like to think of in terms of the martial arts principle, we're using the enemy's characteristics against them.

Because of this exponential blow-up, we have exactly the right characteristics to have a move available at every level as an insurance policy against not getting through to the next level.

And, finally, this whole idea of progressive deepening can be viewed as a prime example of what we like to call anytime algorithms that always have an answer ready to go as soon as an answer is demanded.

So, as soon as that clock runs out at two minutes, some answer is available.

It'll be the best one that the system can compute in the time available given the characteristics of the game tree as it's developed so far.

So, there are other kinds of anytime algorithms.

This is an example of one.

That's how all game playing programs work, minimax, plus alpha-beta, plus progressive deepening.

Christopher, is alpha-beta a alternative to minimax?

CHRISTOPHER: No.

SPEAKER 1: No, it's not.

It's something you layer on top of minimax.

Does alpha-beta give you a different answer from minimax?

CHRISTOPHER: No.

No, it doesn't.

SPEAKER 1: Let's see everybody shake their head one way or the other.

It does not give you an answer different from minimax.

That's right.

It gives you exactly the same answer, not a different answer.

It's a speed-up.

It's not an approximation.

It's a speed-up.

It cuts off lots of the tree.

It's a dead horse principle at work.

You got a question, Christopher?

CHRISTOPHER: Yeah, since all of the lines progressively [INAUDIBLE], is it possible to keep a temporary value if the value [INAUDIBLE] each node of the tree and then [INAUDIBLE]?

SPEAKER 1: Oh, excellent suggestion.

In fact, Christopher has just-- I think, if I can jump ahead a couple steps-- Christopher has reinvented a very important idea.

Progressive deepening not only ensures you have an answer at any time, it actually improves the performance of alpha-beta when you layer alpha-beta on top of it.

Because these values that are calculated at intermediate parts of the tree are used to reorder the nodes under the tree so as to give you maximum alpha-beta cut-off.

I think that's what you said, Christopher.

But if it isn't, we'll talk about your idea after class.

So, this is what every game playing program does.

How is Deep Blue different?

Not much.

So, Deep Blue, as of 1997, did about 200 million static evaluations per second.

And it went down, using alpha-beta, about 14, 15, 16 levels.

So, Deep Blue was minimax, plus alpha-beta, plus progressive deepening, plus a whole lot of parallel computing, plus an opening book, plus special purpose stuff for the end game, plus-- perhaps the most important thing--

uneven tree development.

So far, we've pretended that the tree always goes up in an even way to a fixed level.

But there's no particular reason why that has to be so.

Some situation down at the bottom of the tree may be particularly dynamic.

In the very next move, you might be able to capture the opponent's Queen.

So, in circumstances like that, you want to blow out a little extra search.

So, eventually, you get to the idea that there's no particular reason to have the search go down to a fixed level.

But, instead, you can develop the tree in a way that gives you the most confidence that your backed-up numbers are correct.

That's the most important of these extra flourishes added by Deep Blue when it beat Kasparov in 1997.

And now we can come back and say, well, you understand Deep Blue.

But is this a model of anything that goes on in our own heads?

Is this a model of any kind of human intelligence?

Or is it a different kind of intelligence?

And the answer is mixed, right?

Because we are often in situations where we are playing a game.

We're competing with another manufacturer.

We have to think what the other manufacturer will do in response to what we do down several levels.

On the other hand, is going down 14 levels what human chess players do when they win the world championship?

It doesn't seem, even to them, like that's even a remote possibility.

They have to do something different, because they don't have that kind of computational horsepower.

This is doing computation in the same way that a bulldozer processes gravel.

It's substituting raw power for sophistication.

So, when a human chess master plays the game, they have a great deal of chess knowledge in their head and they recognize patterns.

There are famous experiments, by the way, that demonstrate this in the following way.

Show a chessboard to a chess master and ask them to memorize it.

They're very good at that, as long as it's a legitimate chessboard.

If the pieces are placed randomly, they're no good at it at all.

So, it's very clear that they've developed a repertoire of chess knowledge that makes it possible for them to recognize situations and play the game much more like number 1 up there.

So, Deep Blue is manifesting some kind of intelligence.

But it's not our intelligence.

It's bulldozer intelligence.

So, it's important to understand that kind of intelligence, too.

But it's not necessarily the same kind of intelligence that we have in our own head.

So, that concludes what we're going to do today.

And, as you know, on Wednesday we have a celebration of learning, which is familiar to you if you take a 309.1.

And, therefore, I will see you on Wednesday, all of you, I imagine.